



Department of Electronic Engineering

Design Report

Module: Systems Programming for Embedded Devices

Exam No: Y3900994

Table of Contents

Introduction	4
Specification.....	5
Design.....	6
User Interface Design.....	6
System Design & Peripherals	7
Software Design	8
Software Verification	12
User Manual.....	13
Conclusions	14
References	15

Table of Figures

Figure 1- Screen graphical design	6
Figure 2- Devised buttons and their purpose	6
Figure 3- The overview structure of software, taken from [1]	9
Figure 4- GUI hierarchical state machine diagram	9
Figure 5- Backend task pseudo code	10
Figure 6- Debouncing algorithm pseudocode.....	11
Figure 7- LCD Screen	13
Figure 8- Layout of pushbuttons	13

Introduction

This report aims to give comprehensive information about the final project assignment for Systems Programming for Embedded Devices. In this project, a stopwatch with various functionalities was designed and implemented on ARM cortex M-4 and functional aspects of the system were appropriately tested.

The user interacts with this product using a set of pushbutton available on the board and the output of the device will be displayed on an LCD screen. The product is comprised of two core functionalities named clock mode and stopwatch mode. In the clock mode, the time will be displayed and the user can change the hour display (12/24). In stopwatch mode, the device can measure the passage of time down to the hundredth of a second. The device supports lock mode, where the device enters low-power mode and the screen will get turned off and the user can no longer interact with the device until the user interface is activated again. The design for this project was done with event-driven hierarchical state machines and freeRTOS.

The project started with a given set of specifications and from there the overall system design was decided. In the next step, the design was implemented on the platform using the C language. The software environment used in this project is Keil uVision. In the end, the whole product underwent the testing and verification stage so the functionality of the device could be assured. For each of these stages, detailed information is provided in the following sections of this report.

Specification

Product main functions:

- Display Current time
- Stopwatch

Clock display Facilities:

- Selectable 12/24 hour display
- Display hours/minutes/seconds
- Low power dormant mode

Stopwatch Facilities:

- Start Stopwatch
- Stop Stopwatch
- Clear Stopwatch
- Display hours/minutes/seconds/hundredth seconds
- Low power dormant mode

Design

User Interface Design

When the product is launched, the screen will light up and the following page will show up on the screen:

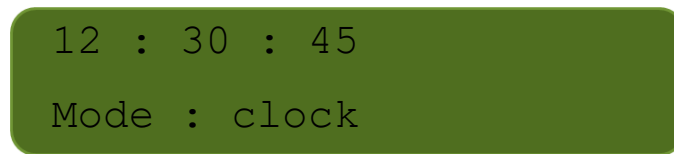


Figure 1- Screen graphical design

The LCD screen available on the board is dedicated to be used as the display.

With the occurrence of any internal or external events, the screen will be updated accordingly. A similar page will show up for stopwatch mode with the addition of a hundredth of a second number. With the following pushbuttons, the user can interact with the product and change the flow of the program:



Figure 2- Devised buttons and their purpose

In clock mode, pressing the “12/24” button results in toggling the hour mode. In stopwatch mode, by pressing the confirm button once, the stopwatch gets started and by pressing that button again, the stopwatch stops. When the stopwatch is stopped, pressing the reset button clears the measured time and re-initiates the time to zero again.

In both modes, pressing the sleep button when the device is active, brings the device to sleep mode where the screen is turned off and user interface is disabled, and repressing the sleep button turns on the screen again.

System Design & Peripherals

Aside from the CPU that utilizes a real-time operating system, few peripherals were set up to realize the targeted functionalities for this product. Three timer peripherals were configured in default up counter mode. One is used for measuring the passage of one second and another one is used for stopwatch mode for measuring a hundredth of a second. The third timer is used to perform polling on the pushbuttons and issues an interrupt every 0.1 seconds. These peripherals will communicate with the CPU using interrupts, so CPU cycles would not be wasted on polling.

One GPIO was utilized to manage the inputs from push buttons. Since the input from pushbuttons is susceptible to the mechanical bouncing effect, the interrupt option for it was not enabled, as it would have sent false signals because of many fluctuations of voltage in mechanical buttons. Instead, the responsibility to read push button inputs was assigned to the CPU side, where proper debouncing function can be implemented efficiently using few lines of code and a small computation load. Another GPIO was configured as output and connected to LEDs for debugging purposes.

The LCD is configured and connected to the system using the provided code script in the labs.

	Purpose	Interrupt priority
Timer3	Measure 1 second	6
Timer4	Measure 0.1 second	8
Timer5	Measure 0.01 second	7
GPIOD	Driving the LEDs connected on board	-
GPIOE	Reading the pushbuttons	-

Table 1- Peripherals and their priorities

Software Design

For designing the software, the event-driven practice was used coupled with hierarchical state machines. In Event-driven programming computation only happens when there is an event happening in the system and it is an antidote to mindless polling loops that waste CPU cycles. This practice is based on the “Inversion of Code” principle where the flow of the control in the software is set and governed by the events, and code will be executed only when an event has occurred. In that sense, it is the most effective and efficient design strategy because CPU cycles wouldn’t be wasted on polling or in an endless super-loop.

Traditional programming techniques use a multitude of variables and flags for saving a history of occurred events in the systems, such practice leads to “Spaghetti code” and makes the codebase hard to edit and maintain, and difficult to read and understand. For avoiding this problem, in this project, hierarchical state machines are used to structure the main functionality of the software program.

The demanded functions from specification are separated into two general tasks, backend and frontend. The tasks which are event-driven only block at the top, waiting for new events and after that the program flows in a run-to-completion manner without any further blocking which is a characteristic of event-driven programming. The frontend task manages the inputs from the user and based on that updates the screen or sends messages to the backend. The backend task takes care of data manipulation and storage and updating core time values. The reason for this division of functions into two tasks is to have them running virtually in parallel, so they wouldn’t be executed sequentially one after another. In this way, we would both have a responsive user interface and accurately keep account of timing registers at the same time. For that reason, these two tasks were given the same priority.

Besides these two tasks, another task is used as a callback function for a 0.1 second timer which has the responsibility of reading inputs from the user (push buttons) and generating appropriate events for other tasks. The task will unblock when a semaphore from ISR is given. This task which runs in a polling-way is used to eliminate the bouncing effect accompanied with the read data from push buttons. Because of the nature of the bouncing effect, interrupts and event-driven practice couldn’t be used to handle these inputs and would have been unreliable. Since this task doesn’t have any hard real-time strict deadlines, it was given a lower priority than the other tasks.

Moreover, to avoid the associated problems with sharing variables between different threads (like race conditions, deadlocks, resource starvation), a “Shared-nothing architecture” has been adopted, every task has its own set of private variables, and communication between tasks only happens with queues and asynchronous messages. In this way, each task runs independently by default unless they really want a result from another task.

The call-back functions from interrupts have been kept short to not disturb the running tasks and they only send messages in the queues or give a semaphore and finish.

Overall, the approximately created software architecture is brought in below:

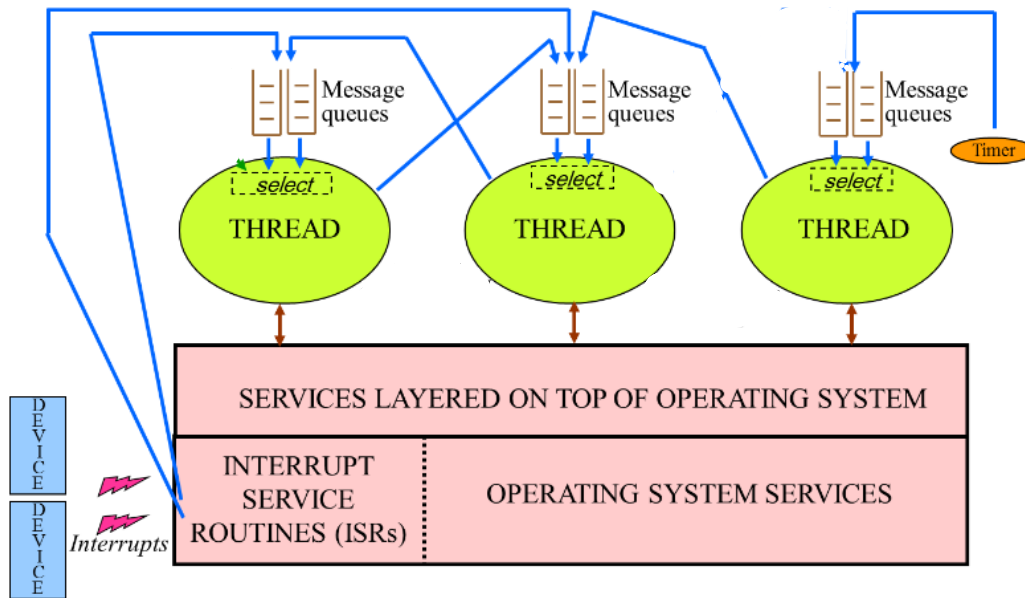


Figure 3- The overview structure of software, taken from [1]

The state machines that delineate how functions from the specification are implemented in the software and how they get related to complete system design are brought in below:

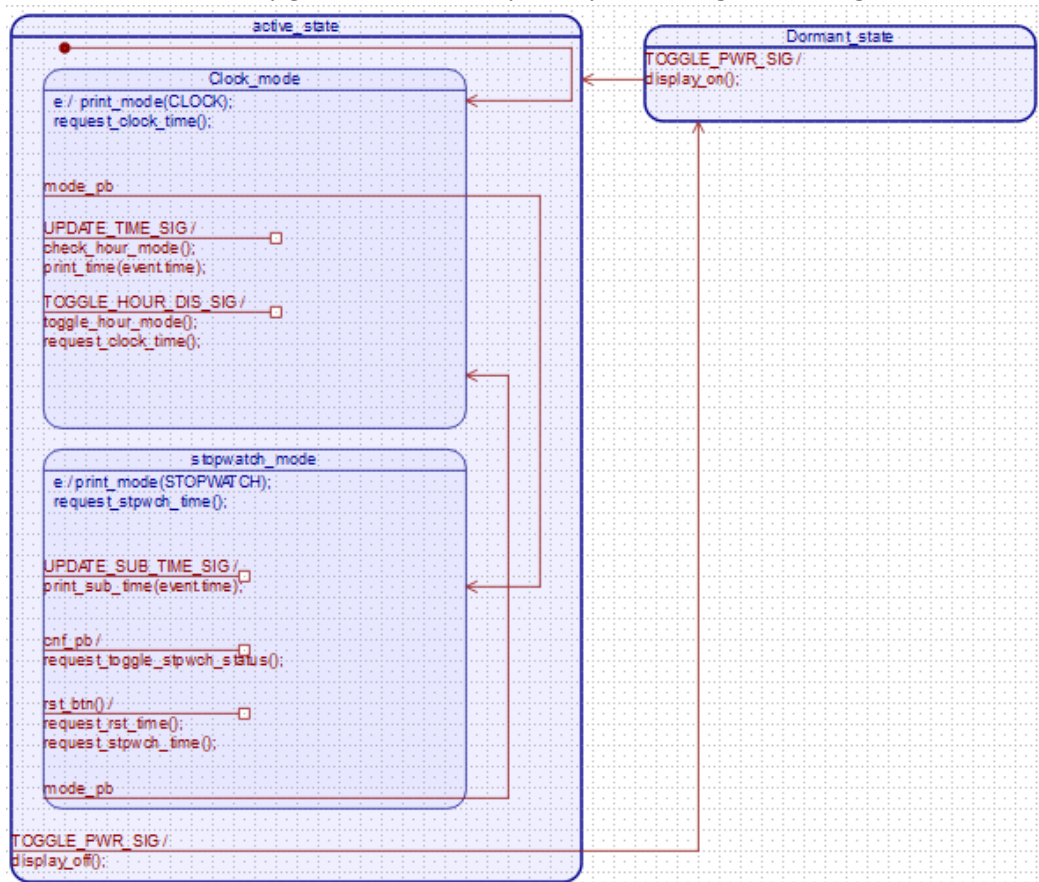


Figure 4- GUI hierarchical state machine diagram

The hierarchical state machine is written in an “inside out” way, the code execution starts from the current most inner state and then in the default case of each switch case, the flow of code would transition to the parent state and passes the received event.

Because of the simplicity of the backend task, the state machines were not used for the design, instead, the design was done using the traditional coding style. The pseudo-code for design is brought in below:

```
switch(time_evt.sig)
{
    case SEC_SIG:
        // All update calculations will be applied to 24 hour mode
        update_time_reg(&clock.time);
        time_to_GUI_evt.sig = UPDATE_TIME_SIG;
        time_to_GUI_evt.time = clock.time;
        xQueueSendToBack ( GUIQueue, &time_to_GUI_evt, portMAX_DELAY);
        break;

    case SUB_SEC_SIG:
        if(stopwatch.status == ENABLED) {
            update_sub_time_reg(&stopwatch.time);
            time_to_GUI_evt.sig = UPDATE_SUB_TIME_SIG;
            time_to_GUI_evt.time = stopwatch.time;
            xQueueSendToBack ( GUIQueue, &time_to_GUI_evt, portMAX_DELAY);
        }
        break;

    case CLOCK_TIME_RQ_SIG:
        time_to_GUI_evt.sig = UPDATE_TIME_SIG;
        time_to_GUI_evt.time = clock.time;
        xQueueSendToBack ( GUIQueue, &time_to_GUI_evt, portMAX_DELAY);
        break;

    case STPWCH_TIME_RQ_SIG:
        time_to_GUI_evt.sig = UPDATE_SUB_TIME_SIG;
        time_to_GUI_evt.time = stopwatch.time;
        xQueueSendToBack ( GUIQueue, &time_to_GUI_evt, portMAX_DELAY);
        break;

    case TGGLE_STPWCH_SIG:
        toggle_stpwtch_status(&stopwatch);
        break;

    case RST_STPWCH_SIG:
        if ( stopwatch.status == DISABLED )
        {
            initialize_time_to_zero(&stopwatch.time);
        }
        break;

    default:
        break;
}
```

Figure 5- Backend task pseudo code

The debouncing algorithm works by detecting the rising edge of GPIO signal by comparing the previous and current value of GPIO buttons. The algorithm is very short and is able to do debouncing for several inputs at the same time in a few lines of code. The pseudo-code for the algorithm is brought in below:

```
    evt send_GUI_evt;
    current = BSP_ReadInput();
    // Next line determines the rising edge of the pb
    risedge = (current & previous) ^ current;

    // If Mode button is pressed
    if ( risedge & BTNC_MASK)
    {
        send_GUI_evt.sig = MODE_PB_SIG;
        xQueueSendToBackFromISR( GUIQueue, &send_GUI_evt, &xHigherPriorityTaskWoken);
    }

    previous = current;
```

Figure 6- Debouncing algorithm pseudocode

Software Verification

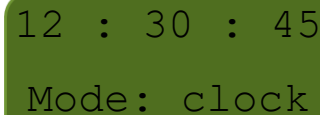
For testing the final product which contains all of the components, the following testbench was designed that is based on providing inputs to the systems in different states and observing the outputs on the screen. The output result of each test vector was according to expectation and is described in the table below.

INPUTS (TEST VECTORS)		STATES	ACTIVE STATE			DORMANT STATE
			CLOCK_ST	STOPWATCH_ST		
				DISABLED	ENABLED	
NO INPUT		NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	
USER INPUTS	MODE PB PRESS	TRANSITION TO STOPWATCH_ST	TRANSITION TO CLOCK_ST	TRANSITION TO CLOCK_ST	NO ACTION (NORMAL OPERATION)	
	CONFIRM PB PRESS	NO ACTION (NORMAL OPERATION)	STOPWATCH STATUS CHANGED TO ENABLED	STOPWATCH STATUS CHANGED TO DISABLED	NO ACTION (NORMAL OPERATION)	
	HOUR MODE PB PRESS	HOUR MODE CHANGED	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	
	SLEEP PB PRESS	TRANSITION TO DORMANT_ST	TRANSITION TO DORMANT_ST	TRANSITION TO DORMANT_ST	TRANSITION TO ACTIVE_ST	
	RESET PB PRESS	NO ACTION (NORMAL OPERATION)	THE TIME ON SCREEN WAS CLEARED TO ZERO	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	
INTERNAL INPUTS	UPDATE TIME SIGNAL	NEW CLOCK TIME UPDATED ON SCREEN	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	
	UPDATE SUB TIME SIGNAL	NO ACTION (NORMAL OPERATION)	NO ACTION (NORMAL OPERATION)	NEW STOPWATCH TIME UPDATED ON SCREEN	NO ACTION (NORMAL OPERATION)	

Table 2- Test vector table

User Manual

To launch the product, first, the microcontroller should be programmed from Keil uVision. After successful booting of the program, the LCD screen located on the board will light up and the following page will appear on the screen:

A green rectangular LCD screen displaying the time '12 : 30 : 45' on the top line and 'Mode: clock' on the bottom line.

12 : 30 : 45
Mode: clock

Figure 7- LCD Screen

The functionalities of the device can be changed using the provided pushbuttons located on the bottom side of the board. The layout and functionality of each button on the board are as follows:



Figure 8- Layout of pushbuttons

The default starting mode of the product is the clock mode. In clock mode, pressing the “12/24” button results in toggling the hour mode from 24 to 12 and vice versa. The starting time on the screen is a default that couldn’t be synced with the real current time since the boards are powered on and off intermittently and are not continuously active. Hence the starting time can only be changed from firmware.

By pressing the “mode” button, you’ll get transferred to stopwatch mode, the stopwatch can get started by pressing the “confirm” button. Once the stopwatch is started, it can be paused again by pressing the same “confirm” button. When the stopwatch is stopped, pressing the reset button clears the measured time and re-initiates the time to zero again. By pressing the “mode” button again, the mode of the product will be transferred to clock mode again.

In both modes, pressing the sleep button when the device is active, brings the device to sleep mode where the screen is turned off and buttons are inactive, and repressing the sleep button turns on the screen again.

Conclusions

In summary, this report presented the details of how an embedded system was developed on an ARM cortex microcontroller and what decisions have been made in each stage. The task started with a set of specifications and from there the design for the product was unfolded, followed by details about, implementation and verification. It should be noted that this is a succinct complementary document for the project and should be read alongside the program source files for complete information.

References

- [1] D. M. Cummings, "Managing Concurrency in Complex Embedded Systems".